

Tool Support for Unit Testing of Aspect-Oriented Software

Jianjun Zhao

Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0295, Japan
zhao@cs.fit.ac.jp

ABSTRACT

In this paper, we present a data flow based unit testing approach and its tool support for aspect-oriented software. Our approach tests two types of units for an aspect-oriented program, i.e., *aspects* that are modular units of crosscutting implementation of the program, and those *classes* whose behaviors may be affected by one or more aspects. We use the control flow graph as a basis to compute def-use pairs of aspects or classes being tested in an aspect-oriented program and use such information to perform data-flow testing on the aspects and classes of the program.

1. INTRODUCTION

Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development [2, 3, 6, 7]. The techniques of AOSD make it possible to modularize crosscutting aspects of a system. Like objects in object-oriented software development, aspects in AOSD may arise at any stage of the software life cycle, including requirements specification, design, implementation, etc. Some examples of crosscutting aspects are exception handling, synchronization, performance optimizations, and resource sharing.

The current research so far in AOSD is focused on problem analysis, software design, and implementation techniques, resulting in a potpourri of representations and procedures. Even though the importance of verification and validation is known, it has commanded little attention in the aspect-oriented paradigm. Although it has been frequently claimed that applying an AOSD method will eventually lead to quality software, aspect-orientation does not provide correctness by itself. An aspect-oriented design can lead to a better system architecture and an aspect-oriented programming language enforces a disciplined coding style, but they are by no means shields against programmer's mistakes or a lack of understanding of the specification. As a result, software testing remains an important task even in AOSD.

Aspect-oriented software differs significantly from procedural and object-oriented software in terms of analysis, design, structure, and development techniques. In aspect-oriented software, the basic unit of organization is the aspect (or class) construct. An aspect with its encapsulation of state with associated advice, introduction, and methods (operations) is a significantly different abstraction in comparison to the procedure or class unit within procedural or object-oriented software. The inclusion of join points in an aspect where piece of code can be advised or introduced to one or more classes further complicates the static and dynamic relationships among aspects and classes. These specific features in aspect-oriented software require special testing support and also provide opportunities for exploitation by a testing strategy. However, although many testing approaches have been proposed for procedural and object-oriented software, they can not be applied to aspect-oriented software. Therefore, new testing techniques and tools that are appropriate for testing aspect-oriented software are strongly needed.

This paper presents a data flow based unit testing approach and its tool support for aspect-oriented software. While unit testing is to test each unit (basic component) of the software to verify that the detailed design for the unit has been correctly implemented [10], data-flow testing is to test how values which are associated with variables can effect the execution of the program [5, 4, 9]. Our approach uses a combination of these two types of testing to test aspects and classes in aspect-oriented software.

In aspect-oriented software, the basic testing unit is an aspect (or class). An aspect (or class) is designed to work as independently as possible from its environment. This is a benefit to unit testing, since it allows the programmer to write a small testing program to exercise the aspect (or class) along. However, on the other hand, an aspect may affect the behavior of one or more classes through advice makes the interactions between the aspect and affected classes more complex. Therefore, when a programmer performs unit testing on a class, the programmer should consider not only the class being tested but also those aspects that may affect the behavior of the class.

Our unit testing approach tests two types of units for an aspect-oriented program, i.e., *aspects* that are modular units of crosscutting implementation of the program, and those *classes* whose behaviors may be affected by one or more aspects. For each aspect or class, our approach performs

three levels of testing, i.e., *intra-module*, *inter-module*, and *intra-aspect/class* testing. For an individual module such as a piece of advice, a piece of introduction, or a method, or a public module along with other modules it calls in an aspect or class, we perform intra-module or inter-module testing. For modules that can be accessed outside the aspect or class and can be invoked in any order by users of the aspect or class, we perform intra-aspect or intra-class testing. While our three-level testing is similar to that of object-oriented programs [5], our approach can handle testing problems unique to aspect-oriented software. We use the control flow graph as a basis for computing def-use pairs of aspects and classes being tested in an aspect-oriented program and use such information to perform data flow testing on the aspects and classes.

The rest of the paper is organized as follows. Section 2 briefly introduces the basic concept of AspectJ. Section 3 discusses some issues that arises in testing aspects and classes in aspect-oriented software, and describes the data-flow testing of aspects and classes. Section 4 discusses tool support for our testing approach. Concluding remarks are given in Section 5.

2. ASPECTJ

In this paper, we will use AspectJ as our target language to show the basic idea of unit testing for aspect-oriented software.

AspectJ [1] is a seamless aspect-oriented extension to Java. AspectJ adds some new concepts and associated constructs to Java. These concepts and associated constructs are called join points, pointcuts, advice, and aspect.

The *join point* is essential element in the design of any aspect-oriented programming language since join points are the common frame of reference that defines the structure of crosscutting concerns. The join point in AspectJ are well-defined points in the execution of a program. A *pointcut* is a set of join points that optionally exposes some of the values in the execution of these join points. AspectJ defines several primitive *pointcut designators* that can identify all types of join points. Pointcuts in AspectJ can be composed and new pointcut designators can be defined according to these combinations.

Advice is a method-like mechanism used to define certain code that is executed when a pointcut is reached. There are three types of advice, that is, *before*, *after*, and *around*. In addition, there are also two special cases of after advice, *after returning* and *after throwing*, corresponding to the two ways a sub-computation can return through a join point.

Aspects are modular units of crosscutting implementation. Aspects are defined by aspect declarations, which have a similar form of class declarations. Aspect declarations may include pointcut declarations, advice declarations, as well as other declarations such as method declarations, that are permitted in class declarations.

An AspectJ program is composed of two parts: (1) *non-aspect code* which includes some classes, interfaces, and other language constructs as in Java, (2) *aspect code* which in-

cludes aspects for modeling crosscutting concerns in the program. Moreover, any implementation of AspectJ is to ensure that aspect and non-aspect code run together in a properly coordination fashion. Such a process is called *aspect weaving* and involves making sure that applicable advice runs at the appropriate join points. A sample AspectJ program taken from [1] is shown below. For more information about AspectJ, one can refer to [1]. In the rest of the paper, for unification, we use the word “a module” to stand for a piece of advice, a piece of introduction, or a method declared in an aspect, and a method declared in a class.

```
class Point {
    int x, y;
    public void setX(int x) {
        this.x = x;
    }
    public void setY(int y) {
        this.y = y;
    }
}
class Line {
    private Point p1, p2;
    public void setP1(Point p1) {
        this.p1 = p1;
    }
    public void setP2(Point p2) {
        this.p2 = p2;
    }
}
aspect PointBoundsPreCondition {
    before(int x): call(void Point.setX(int)) && args(x) {
        if ( x < MIN_X || x > MAX_X )
            throw new RuntimeException();
    }
}
```

3. DATA FLOW TESTING FOR ASPECTS AND CLASSES

In this paper, we consider data-flow based unit testing of aspects and classes in aspect-oriented software to verify that the detailed design for aspects and classes has been correctly implemented.

An AspectJ program is composed of some aspects, classes, and interfaces. In this paper we only consider testing of aspects and classes, leaving interfaces as our future work. Aspects are modular units that implement crosscutting concerns in aspect-oriented software and are different from classes in object-oriented software. Classes in aspect-oriented software are similar to those in object-oriented software, but have some differences compared to those in aspect-oriented software. This is because that a class' behavior in an aspect-oriented program may be affected by one or more aspect(s). In order to identify the problems of testing aspects and classes of aspect-oriented programs, we present a motivation example to discuss the issues that arise in testing aspects and classes of an aspect-oriented program.

3.1 Motivation Example

Consider an aspect, `PointBondsPreCondition` shown below, that declares a piece of before-advice to modify the behavior of class `Point`'s `setX` method. The before-advice can be applied to each join point where a target object of type `Point` receives a call to its method with signature

`void Point.setX(int)`. The `args` keyword is used to assign the argument of the method call.

In AspectJ this before-advice is applied by the compiler without explicit reference to the aspect from either the `Point` module or a client module. So existing unit testing of class `Point` does not consider the `PointMoveChecking` aspect, and therefore has no way to know that the behavior of the `setX` method may be changed when class `Point` and aspect `PointBoundsPreCondition` are compiled together. However, when they are compiled together, then intuitively the behavior of `Point`'s `setX` method has to be changed due to the before-advice. Thus, in order to perform unit testing on class `Point`, we should take into account not only the `Point` itself but also aspect `PointBoundsPreCondition` that affects the behavior of `Point` through the before-advice.

In order to efficiently identify those classes whose behavior may be affected by one or more aspects, we classify classes in an aspect-oriented program into two categories: *variable classes* whose behavior may be affected by one or more aspects and *stable classes*, otherwise. In this paper, we do not consider a stable class, since we can perform data-flow testing on it by using existing techniques in [5]. For a variable class, since there exists no unit testing technique that can be applied to it, it is our main focus in this paper. As an example, consider the program shown previously, which is composed of one aspect `PointBoundsPreCondition` and two classes `Point` and `Line`. From the definition given above, class `Point` is a variable class because the behaviors of the method `setX` may be changed by the before-advice declared in aspect `PointBoundsPreCondition`. On the other hand, class `Line` is a stable class since no aspect in the program affects its behavior.

3.2 Data Flow Testing

In this section, we first describe three-level testing for aspects and classes of aspect-oriented, and then show how to construct the (framed) control flow graphs for an aspect or class, and how to compute three kinds of def-use pairs for the aspect or class being tested based on these graphs.

3.2.1 Three Level Testing for Aspects and Classes

When we perform unit testing on aspects and classes, we also consider three different levels of testing, i.e., *intra-module*, *inter-module*, and *intra-aspect/class* testing. Our three-level testing is similar to that of object-oriented software [5], but applied to handle testing problems unique to aspect-oriented software. There is an important issue that must be considered during unit testing of aspects or classes, that is, the behavior of a method being tested in a class may be affected (changed) by one or more pieces of advice in an aspect. To cover this issue, we classify methods of a class into two categories: *variable methods* whose behavior may be affected (changed) by one or more pieces of advice in an aspect, and *normal methods*, otherwise. For normal methods, we need not treat them specially. However, for a variable method, since its behavior may be changed by some advice, we should consider it together with its advised advice when performing three-level testing on them.

Intra-Module Testing. Intra-module testing means performing testing on an individual module such as a piece

of advice, a piece of introduction, or a method in an aspect or class. Therefore intra-module testing of an aspect or class includes three forms of testing: *intra-advice*, *intra-introduction*, and *intra-method* testing.

Inter-Module Testing. Inter-module testing means performing testing on a public module along with some other modules it calls, directly or indirectly, in an aspect or class. Intra-module testing does not consider invocations from other modules outside the aspect or class. More concretely, intra-module testing aims at testing internal interactions among modules within an aspect or class.

Intra-Aspect/Class Testing. Intra-aspect/class testing means performing testing on the interactions of multiple public modules in an aspect or class when they are called in a random sequence from the outside of the aspect or class. Unlike inter-module testing that only considers one public module along with other modules it calls within an aspect or class, intra-aspect testing considers multiple modules and their interactions in an aspect or class, allowing multiple calls to these modules from the outside of the aspect or class.

3.2.2 Computing Def-Use Pairs for Aspects and Classes

Our data-flow unit testing for an aspect or class considers all aspect or class variables and def-use pairs that are closed related to some specific program points in the aspect or class. We identify three types of def-use pairs, i.e., *intra-module*, *inter-module*, and *intra-aspect/class* def-use pairs, in an aspect or class that should be tested to correspond to the three-level testing of the aspect or class described above. Similar to [5], we use the algorithm presented in [8] to compute the intra-module and inter-module def-use pairs. For intra-aspect/class def-use pairs we can borrow the idea from [5] to use a similar representation called the framed control-flow graph to represent an aspect or class that can be used as a base to compute intra-aspect/class def-use pairs.

We construct the framed control-flow graph for an aspect or class to compute intra-aspect/class def-use pairs of the aspect or class. The graph can simulate a random calling sequence between modules in an aspect or class, and therefore support for performing data-flow analysis on the aspect or class. A *framed control-flow graph* (FCFG) of an aspect or class consists of a collection of CFGs each presents a module in the aspect or class and some additional arcs used to construct the frame. In an FCFG, there are some vertices used to represent the frame such as *frame entry vertex*, *frame loop*, *frame call*, *frame return*, and *frame exit*. The frame call vertex is connected to the *entry vertex* of each CFG for modules. If there is a call in a module to call another module in an aspect or class, we connect two modules' CFGs at call sites using *call arcs*.

The FCFG for an aspect or class can be constructed by the following three steps: (1) First, we construct an aspect or class call graph to represent the call relationships among modules in an aspect or class. (2) Second, we construct a frame to represent a test driver for the aspect or class and enclose it into the aspect or class call graph to form a partial FCFG. The frame allows one to simulate a random calling

sequence to some modules in an aspect or class, (3) Third, we replace each vertex v of the aspect or class call graph in the partial FCFG with the control-flow graph for v .

A *control-flow graph* (CFG) for a module M of an aspect or class represents the static control flow relationships that exist within M of the aspect or class. The CFG of M contains a vertex for each statement in M and arcs between vertices that represent flow of control between statements. There is also a unique vertex called *entry vertex* to represent the unique entry to M , and a unique vertex called *exit vertex* to represent exit from M .

A *call graph* for an aspect or class is a digraph such that its vertices represent modules and its arcs represent calling relations between modules in the aspect or class. We use the call graph to represent the call structure of an aspect or class which can be used to aid construction of the FCFG.

4. TOOL SUPPORT

To perform data-flow based unit testing on aspects and classes for aspect-oriented software, we need both control flow and data flow information for each aspect or class being tested. We plan to implement a unit testing tool for AspectJ to realize the approach presented in this paper. Our tool has three components, the *driver generator*, the *compiler*, and the *test case generator*.

The driver generator takes as input some related aspects and classes and outputs a test driver. This test driver, when executed, reads test cases, check their syntax, executes them, and checks the results. As the first step, currently we generate our test driver by hand.

The compiler takes as input an AspectJ program, and analyze the program to get control flow information such as the predecessors and successors of each statement and caller-callee information, and data flow information such as definition and use of each variable in each statement, which are necessary for constructing control flow graphs and computing def-use pairs for each module, group of modules, and aspect or class. Based on such information the compiler first constructs the control flow graph for each module in an aspect or class, and then constructs the call graph for each aspect or class in the program. Finally, it constructs the framed control flow graph for each aspect or class in the program. It also computes def-use pairs for each module, group of modules, and aspect or class in the program.

The test case generator takes as input def-use pairs information outputted by the compiler component, and generates various test cases for each module, group of modules, and aspect or class being tested, which are used by the test driver.

We plan to modify the AspectJ compiler (`ajc`) for gathering the control flow and data flow information of an AspectJ program for our purpose.

5. CONCLUDING REMARKS

In this paper, we presented a data flow based unit testing approach and its tool support for aspect-oriented software. Our approach tests two types of units for an aspect-oriented

program, i.e., *aspects* that are modular units of crosscutting implementation of the program, and those *classes* whose behaviors may be affected by one or more aspects. While our three levels of testing is similar to that of object-oriented software, our approach can handle testing problems unique to aspect-oriented software. We used the control flow graph as a basis for computing def-use pairs of aspects or classes being tested in an aspect-oriented program and used such information to perform data flow testing for the aspects and classes.

Future work may include (1) to present more precise technique to compute various def-use pairs for an aspect or class and use such information to refine our data-flow unit testing approach for aspects and classes of aspect-oriented software, and (2) to implement a data-flow based unit testing tool for AspectJ to realize the approach presented in this paper.

6. REFERENCES

- [1] The AspectJ Team, "The AspectJ Programming Guide," August 2001. <http://aspectj.org>
- [2] L. Bergmans and M. Aksits, "Composing crosscutting Concerns Using Composition Filters," *Communications of the ACM*, Vol.44, No.10, pp.51-57, October 2001.
- [3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *proc. 11th European Conference on Object-Oriented Programming*, pp220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.
- [4] J. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Transaction on Software Engineering*, Vol.9, pp.33-43, May 1983.
- [5] M. J. Harrold and G. Rothermel, "Performing Data Flow Testing on Classes," *Proc. ACM SIGSOFT Foundation of Software Engineering*, 1994.
- [6] K. Lieberher, D. Orleans, and J. Ovlinger, "Aspect-Oriented Programming with Adaptive Methods," *Communications of the ACM*, Vol.44, No.10, pp.39-41, October 2001.
- [7] H. Ossher and P. Tarr, "Multi-Dimensional Separation of Concerns and the Hyperspace Approach," *Proc. the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer, 2001.
- [8] H. Pande, W. Landi, and B. G. Ryder, "Interprocedural Def-Use Associations in C Programs," *IEEE Transaction on Software Engineering*, Vol.20, No.5, pp.385-403, May 1994.
- [9] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transaction on Software Engineering*, Vol.11, No.4, pp.367-375, April 1985.
- [10] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, Vol.29, No.4, pp.366-427, December 1997.